JOURNAL OF SOFTWARE MAINTENANCE: RESEARCH AND PRACTICE
J. Softw. Maint: Res. Pract. **11**, 1–18 (1999)

Practice

Which Software Modules have Faults which will be Discovered by Customers?

TAGHI M. KHOSHGOFTAAR[1]*, EDWARD B. ALLEN[1], WENDELL D. JONES[2] and JOHN P. HUDEPOHL[2]
[1]*Department of Computer Science and Engineering, Florida Atlantic University, Boca Raton FL 33431, U.S.A.*
[2]*Software Reliability Engineering Department, Nortel, Research Triangle Park NC 27709, U.S.A.*

SUMMARY

Software is the medium for implementing increasingly sophisticated features during the maintenance phase as successive releases are developed. Software quality models can predict which software modules are likely to have faults that will be discovered by customers. Such models are key components of a system such as Enhanced Measurement for Early Risk Assessment of Latent Defects (EMERALD). It is a sophisticated system of decision support tools used by software designers and managers at Nortel to assess risk and improve software quality and reliability of legacy software systems. This paper reports an approach to software quality modelling that is suitable for industrial systems such as EMERALD.

We conducted a case study of a large legacy telecommunications system in the maintenance phase to predict whether each module will be considered fault-prone. The case study is distinctive in the following respects. (1) Fault-prone modules were defined in terms of faults discovered by customers, which represent only a small fraction of the modules in the system. (2) We developed models based on software product and process metrics that can make useful predictions at the end of the coding phase and at the time of release. (3) The modelling approach is suitable for very large systems. We anticipate that refinements of this case study's models will be incorporated into EMERALD. A similar approach could be taken for other systems. Copyright © 1999 John Wiley & Sons, Ltd.

KEY WORDS: software quality; software metrics; product metrics; process metrics; fault-prone modules; logistic regression

1. INTRODUCTION

Reliability has become a key competitive factor in many industries. For example, society has become so accustomed to reliable telecommunications, that failures cause major disruptions. Accordingly, Bellcore is requiring progressively higher reliability on behalf of operating telephone companies. Development of reliable software in legacy systems is an important element of system reliability, because software is the medium for implementing increasingly sophisticated features (Hudepohl *et al.*, 1992). Consequently, software reliability is becoming a strategic business weapon in an increasingly competitive market-place (Hudepohl, 1990).

*Correspondence to: Taghi M. Khoshgoftaar, Department of Computer Science and Engineering, Florida Atlantic University, Box 3091, Boca Raton FL 33431–0091, U.S.A. Email: taghi@cse.fau.edu
Contract/grant sponsor: Nortel

CCC 1040–550X/99/010001–18$17.50

Received 27 April 1998
Revised 17 September 1998

A *fault* is a defect in source code that causes an operational or test failure. Models based on software metrics can predict which modules are fault-prone, the number of faults expected, or the probability of at least one fault. Such predictions have several benefits. For example, inspections are a standard part of our development process (Russell, 1991). Rather than inspecting all modules on an equal basis, focusing on high-risk modules can improve the efficiency of inspection efforts. Predictions are also useful for planning maintenance activities during the operations phase, and for prioritizing re-engineering efforts during the next maintenance cycle (Briand and Basili, 1992; Coleman, Lowther and Oman, 1995).

Enhanced Measurement for Early Risk Assessment of Latent Defects (EMERALD) is a sophisticated system of decision support tools used by software designers and managers to assess risk and improve software quality and reliability. It was developed by Nortel (Northern Telecom) in partnership with Bell Canada and others (Hudepohl *et al.*, 1996). Lyu *et al.* (1995) reported on a prototype with similar objectives. EMERALD provides access to metrics about the source code, problems related to the source code, source-code changes and deployment usage. It also provides access to risk models based on the metrics. At various points in the maintenance process, EMERALD predicts which modules are likely to be fault-prone. For example, at the time of a high-level design review, suppose EMERALD indicates that certain modules are at risk if changed. After a review, the design team may choose to re-engineer these modules first before continuing implementation of the next release. Risk assessments can also be used for assigning technical staff to test, test automation and maintenance efforts, matching skill level and experience with complexity. Moreover, during maintenance, any changes proposed for high-risk modules could trigger more stringent justification and inspection. The goal of this research is more accurate and robust software quality models that are suitable for systems such as EMERALD. A similar approach can be adopted for other systems.

The contribution of this paper is an industrial case study of a very large telecommunications system that provides empirical evidence of the following:

- Modules which will likely have faults discovered by customers can be identified with useful accuracy prior to release. In our case study, less than eight per cent of changed modules and less than two per cent overall had faults discovered by customers. Others have indicated that this small set of modules can be difficult to identify in advance. Ebert studied faults from the field in a telecommunications system (Ebert, 1997). Schneidewind's fault data were collected from Space Shuttle flight simulator experience, which is almost equivalent to 'customer-discovered' faults (Schneidewind, 1995). Henry and Wake (1991) studied changes to source code after release due to bug fixes. Their study did not specify who discovered the bugs. However, most software quality models in the literature have counted faults discovered during testing, rather than those discovered by customers.
- Models with very useful accuracy were developed for two points in the development life cycle: (1) when coding is complete, and (2) at the completion of beta testing. The former were based on software product metrics, and the latter were based on product and process metrics. The set of process metrics in the case study was more extensive than that used in previous research (Khoshgoftaar *et al.*, 1996a, 1998).
- The case study examined a very large legacy telecommunications system with stringent reliability requirements, demonstrating that our approach is not limited to small systems. We anticipate that a refinement of the case study's models will be incorporated into EMERALD.

The case study's modelling technique was logistic regression (Hosmer and Lemeshow, 1989; Khoshgoftaar and Allen, 1997a, c) in conjunction with a generalized classification rule which we have proposed (Khoshgoftaar and Allen, 1997d). The remaining sections present our modelling methodology, a system description of the case study, modelling results and conclusions. Appendices present details on logistic regression and our classification rule.

2. MODELLING

2.1. Methodology

Our modelling methodology has the following steps (Khoshgoftaar *et al.*, 1996b).

1. Collect data from the configuration management system and problem reporting system.
2. Select modules for analysis. In software quality modelling, we usually define an 'observation' to be a module.
3. Measure *faults* and independent variables. This study considered only faults discovered by customers after release of the software to production.
4. Determine the class of each module:

$$Class = \left\{ \begin{array}{cc} \textit{not fault-prone} & \text{if } \textit{faults} < \textit{threshold} \\ \textit{fault-prone} & \text{if } \textit{fault} \geq \textit{threshold} \end{array} \right\} \quad (1)$$

   where *threshold* is chosen according to project-specific criteria. Our case study's threshold was 1.
5. Prepare *fit* and *test* data sets. A *fit* data set is used to 'fit' a model—i.e., estimate its parameters. A *test* data set is used to evaluate the accuracy of the corresponding fitted model. The set of all observations is used as both a *fit* data set and a *test* data set for the 'best models'. Accuracy is indicated by resubstitution of the data into the model. Data splitting is our primary method for validating models. Derive *fit* and *test* data sets from the data by impartially dividing all modules studied into two subsets several times. In the case study, the fit and test data sets had nearly equal numbers of observations. This assured sufficient observations in each set for the statistical techniques used here. Other proportions might be appropriate in other situations.
6. Select significant independent variables by applying stepwise logistic regression based on the *fit* data set. We used a significance level of $\alpha = 15\%$, which is often used by statisticians with stepwise logistic regression and is the default supplied by the SAS statistical package. See Appendix A for details on stepwise logistic regression.
7. Estimate parameters of the final logistic regression model using the *fit* data set. Logistic regression models are easily and efficiently implemented in a system such as EMERALD (Hudepohl *et al.*, 1996), and are easier to interpret than some competing classification techniques (Khoshgoftaar and Allen, 1997a). Appendix A provides details on logistic regression.
8. Select a value for the classification rule's parameter, so that type I and type II misclassification rates are approximately equal, when resubstituting the *fit* data set into the model. A classification model is hardly ever perfect. Type I errors misclassify modules that are actually

not fault-prone as *fault-prone*. Type II errors misclassify modules that are actually *fault-prone* as *not fault-prone*.

9. Predict dependent variable values for evaluation. Re-substitute the *fit* data set into the model to predict the class of each module, and similarly, apply the model to the *test* data set. Appendix B provides details on the classification rule.
10. Compare the predicted class with the actual class of each module. Calculate misclassification rates, effectiveness and efficiency. *Effectiveness* is defined as the proportion of *fault-prone* modules correctly identified by the model out of all *fault-prone* modules. *Efficiency* is defined as the proportion of *fault-prone* modules correctly identified out of all modules the model says are *fault-prone*.
11. Evaluate the accuracy in the context of using predictions to guide quality improvement activities. The accuracy of re-substitution indicates the quality of fit. This may be a biased estimate of model accuracy, because the *fit* data set is used both for estimating parameters and evaluating accuracy. Therefore, we focus on the accuracy of the *test* data set results, which indicate the predictive ability of the model. Validation tells us the level of accuracy to expect when applying the 'best model' to a subsequent release or a similar project when the actual class membership is not known.

3. CASE STUDY

3.1. System description

The system under study was a very large legacy telecommunications system, written in a high-level language, Protel, and maintained by professional programmers in a large organization. The entire system had significantly more than 10 million lines of code. This embedded computer application included numerous finite-state machines and interfaces to other kinds of equipment.

This case study focused on faults discovered by customers, *faults*, as the software quality metric. A module was considered *fault-prone* if any faults were discovered by customers.

$$Class = \left\{ \begin{array}{cc} \textit{not fault-prone} & \text{if } \textit{faults} = 0 \\ \textit{fault-prone} & \text{otherwise} \end{array} \right\} \tag{2}$$

Fault data were collected at the module level by the problem reporting system. A *module* consisted of a set of related source-code files. Analysis of configuration management data indicated whether or not each module was unchanged from the prior release. Approximately 99 per cent of the unchanged modules had no faults.

This case study considered only new and modified modules. These modules had several million lines of code in a few thousand modules. The proportion of modules with no faults among the updated modules was $\pi_1 = 0.9260$, and the proportion with at least one fault was $\pi_2 = 0.0740$.

USAGE was calculated using data derived from deployment records of an earlier release. Considering that this is a legacy system, *USAGE* was a forecast of usage in the current release. Other software product metrics were collected from source code by the EMERALD system. EMERALD measures attributes of call graphs, control flow-graphs and source code statements (Mayrand and Coallier, 1996; Robillard, Coupal and Coallier, 1991). A call graph depicts invocation relationships

among procedures. A control flow-graph shows the flow of control where each arc represents a series of in-line statements and each node represents a decision statement, such as `IF`, `FOR` and `WHILE` statements, or a branch destination. Product measurements were collected at the procedure level, and then aggregated to the module level. Unless otherwise noted, metrics were aggregated to the module level as an unweighted summation.

Development process data were derived from the configuration management system and problem reporting system, both of which interface with the EMERALD system. Table 1 describes the data used in this study.

3.2. Data set creation

Three times we randomly divided the available data on updated modules into two subsets, creating three pairs of approximately equal *fit* and *test* data sets. Each pair encompassed all available data. This technique is called 'resampling' (Dillon and Goldstein, 1984, p. 392).

3.3. Results from models based on product metrics

For the models in this section, all 25 product metrics listed in Table 1 were candidate independent variables. Stepwise logistic regression selected significant variables. The following model was estimated based on all updated modules. Variables are in order of significance.

$$\log\left(\frac{\hat{p}}{1-\hat{p}}\right) = -5.13 + 0.0164\mathit{FILINCUQ} + 0.0209\mathit{LGPATH} + 1.2718\mathit{USAGE} + 0.00043\mathit{VARSPNMX} \quad (3)$$

where $\hat{p}$ is the probability that the module is *fault-prone*. All variables had a significance level of $\alpha < 1\%$. This model may be used to make predictions as soon as modules have been coded, and as soon as *USAGE* can be estimated from an earlier release and from plans for the current release. Since its parameters were estimated with all the available data, we consider it the 'best model' in this section. It is not necessarily the absolutely best model. Table 2 gives details of the estimated parameters for this model.

We chose a value of c for the generalized classification rule summarized in Appendix B, where resubstitution misclassification rates were approximately equal; we also prefer that the type II misclassification rate be less than the type I rate. This value of $c = 14.38 = (0.0740/0.9260)(180)$ corresponds to prior probabilities based on the proportions of each class, $\pi_2/\pi_1 = 0.0740/0.9260$, and a ratio of costs of misclassification of $C_{\mathrm{II}}/C_{\mathrm{I}} = 180$. Thus, this rule would minimize the expected cost of misclassifications if the value of a lost opportunity due to a type II misclassification was 180 times the direct cost of a wasted reliability enhancement—i.e., the cost of a type I misclassification. This is a plausible order of magnitude, because faults discovered by customers are extremely costly for telecommunications systems.

Table 3 shows details of the accuracy of the best model in this section when resubstituting all available data into the model to make predictions. The type I misclassification rate of the best model in this section was 29.80 per cent and the type II misclassification rate was 25.56 per cent. This translated into an effectiveness of 74.44 per cent and an efficiency of 16.64 per cent. For example,

Table 1. Data elements

Symbol description

Software quality metric

Faults: the number of field problems found in the module against the current issue.

Deployment metric

USAGE: a measure of the deployment percentage of the module. Higher values imply more widely deployed and used.

Call graph metrics

CALUNQ: the number of distinct calls to others.

CAL2: the number of second and following calls to others $CAL2 = CAL - CALUNQ$ where *CAL* is the total number of calls.

Control flow-graph metrics

CNDNOT: the number of arcs that are not conditional arcs.

IFTH: the number of non-loop conditional arcs, i.e., `if-then` constructs.

LOP: the number of loop constructs.

CNDSPNSM: the total span of branches of conditional arcs. The unit of measure is arcs.

CNDSPNMX: the maximum span of branches of conditional arcs. Aggregated to module level by maximum.

CTRNSTSM: the total control structure nesting.

CTRNSTMX: the maximum control structure nesting. Aggregated to module level by maximum.

KNT: the number of knots. A 'knot' in a control-flow graph is where arcs cross due to a violation of structured programming principles.

NDSINT: the number of internal nodes. $NDSINT = NDS - NDSENT - NDSEXT - NDSPND$ where NDS is the total number of nodes.

NDSENT: the number of entry nodes.

NDSEXT: the number of exit nodes.

NDSPND: the number of pending nodes, i.e., dead code segments.

LGPATH: the base 2 logarithm of the number of independent path, *PTHIND*, which is aggregated to the module level by summation. $LGPATH = \log_2 PTHIND$.

Statement metrics

FILINCUQ: the number of distinct include files.

LOC: the number of lines of code.

STMCTL: the number of control statements.

STMDEC: the number of declarative statements.

STMEXE: the number of executable statements.

VARGLBUS: the number of global variables used.

VARSPNMX: the maximum span of variables.

VARUSDUQ: the number of distinct variables used.

Process metrics

CUST_FIX: total number of different problems that were fixed during current maintenance cycle where the problems originated from issues found by customers.

BETA_FIX: total number of different problems that were fixed during current maintenance cycle where the problems originated from issues found by beta testing.

BETA_PR: the number of problems found in this module during beta testing of the current release.

SRC_GRO: net increase in lines of code due to software changes.

UNQ_DES: number of different designers who updated this module.

UPD_CAR: the total number of updates that designers had in their company careers when they updated this module.

VLO_UPD: number of updates to this module by designers who had 10 or less total updates in entire company career.

Table 2. Best model based on product metrics

Variable	All observations	
	Coefficient	Standard deviation
Intercept	−5.13	0.24
FILINCUQ	+0.0284	0.0033
LGPATH	+0.0209	0.0060
USAGE	+1.2718	0.2427
VARSPNMX	+0.00043	0.00013

Other product metrics were not significant at 15 per cent level.

Table 3. Evaluation of product metrics best model, $c = 14.38$. Validation by resubstitution of all observations

Group (actual class)	Predicted by model		Total
	not fault-prone	*fault-prone*	
not fault-prone	70.20%	29.80%	100.0%
fault-prone	25.56%	74.44%	100.0%
Total split	66.90%	33.10%	100.0%

Overall misclassification rate: 29.49%; effectiveness = 74.44%; efficiency = 16.64%.

Table 4. Models based on product metrics

Variable	fit_1		fit_2		fit_3	
	Coefficient	Standard deviation	Coefficient	Standard deviation	Coefficient	Standard deviation
Intercept	−5.15	0.34	−5.09	0.33	−5.69	0.36
FILINCUQ	+0.0310	0.0043	+0.0282	0.0046	+0.0395	0.0046
LGPATH	+0.0227	0.0078	+0.0177	0.0082	+0.0344	0.0078
USAGE	+1.3734	0.3445	+1.2541	0.3364	+1.2628	0.3624
VARSPNMX	—	—	+0.00050	0.00018	—	—

— means variable was not significant at 15 per cent level.

following model recommendations, code reviews would examine three-quarters of the fault-prone modules, but only one out of six reviews would be productive.

Validation using the three *fit* and *test* data-set pairs indicated the variation due to data splitting. Table 4 shows details of the estimated parameters of each model. Each was based on a *fit* data set. Table 5 shows how well the product metrics-based models were able to classify each pair of the *fit* and *test* data sets. For example, the rates for $test_1$ were a type I misclassification rate of 32.62 per cent and a type II misclassification rate of 28.15 per cent. This translated to an effectiveness of 71.85 per cent and an efficiency of 14.97 per cent. These results indicate that one can expect similar

Table 5. Evaluation of product metrics models, $c = 14.38$. Validation by data splitting. Misclassification rates

Data		*fit*		*test*	
		Type I	Type II	Type I	Type II
All observations		29.80%	25.56%		
Data set	1	31.42%	25.19%	32.62%	28.15%
Data set	2	29.96%	26.47%	29.53%	24.63%
Data set	3	26.88%	22.79%	25.03%	39.55%

Data set	Effectiveness	Efficiency
$test_1$	71.85%	14.97%
$test_2$	75.37%	16.94%
$test_3$	60.45%	16.18%

accuracy when classifying modules from a subsequent release or a similar project using a product metrics-based model. We consider the accuracy good, in view of the fact that *fault-prone* modules were rare, less than 8 per cent of those updated.

3.4. Results from models based on product and process metrics

We investigated models of updated modules where predictions could be made at the time of release, i.e. after beta testing. The candidate variables consisted of the four product metrics that were significant in Section 3.3, plus the seven process metrics in Table 1. Stepwise logistic regression selected significant variables. The following model was estimated based on all the updated modules. Variables are in order of significance.

$$\begin{aligned}\log\left(\frac{\hat{p}}{1-\hat{p}}\right) = {} & -4.9932 + 0.2475\mathit{UNQ\_DES} + 0.0164\mathit{FILINCUQ} \\ & + 1.0793\mathit{USAGE} + 0.7342\mathit{BETA\_PR} \\ & + 0.0175\mathit{LGPATH} - 0.2145\mathit{VLO\_UPD} \\ & + 0.2999\mathit{BETA\_FIX} + 0.2461\mathit{CUST\_FIX} \\ & + 0.000197\mathit{SRC\_GRO} + 0.000233\mathit{VARSPNMX} \end{aligned} \quad (4)$$

where $\hat{p}$ is the probability that a module is *fault-prone*. The significance level of *VARSPNMX* was $\alpha = 11.6$ per cent; the level of *SRC_GRO* was $\alpha = 8.0$ per cent; and the significance of each of the other variables was $\alpha < 5$ per cent. Since its parameters were estimated with all the available data, we consider it the 'best model' in this section. Table 6 gives details on the estimated parameters. Like the product metrics-based models, we chose a value of c for the generalized classification rule presented in Appendix B, where resubstitution misclassification rates were approximately equal. This value of $c = 15.98 = (0.0740/0.9260)(200)$ corresponds to a rule that would minimize the expected cost of misclassifications, if we set the value of a lost opportunity due to a type II misclassification at 200 times the direct cost of a wasted reliability enhancement, i.e., the cost of

Table 6. Best model based on product and process metrics

Variable	All observations	
	Coefficient	Standard deviation
Intercept	−4.9932	0.2418
FILINCUQ	+0.0164	0.0037
LGPATH	+0.0175	0.0065
VARSPNMX	+0.000233	0.000148
USAGE	+1.0793	0.2455
UNQ_DES	+0.2475	0.0389
BETA_PR	+0.7342	0.1841
BETA_FIX	+0.2999	0.1430
CUST_FIX	+0.2461	0.1262
VLO_UPD	+0.2145	0.0839
SRC_GRO	+0.000197	0.000129
UPD_CAR	—	—

Table 7. Evaluation of best model based on product and process metrics, $c = 15.98$. Validation by resubstitution of all observations

Group (actual class)	Predicted by model		Total
	not fault-prone	*fault-prone*	
not fault-prone	71.91%	28.09%	100.0%
fault-prone	22.22%	77.78%	100.0%
Total split	68.24%	31.76%	100.0%

Overall misclassification rate: 27.65%; effectiveness = 77.78%; efficiency = 18.12%.

a type I misclassification. We balanced this model at a different value of c ($c = 15.98$) from the $c = 14.38$ at which we had balanced the product metrics-based model.

Table 7 shows details of the accuracy of the best model in this section when resubstituting all available data into the model to make predictions. The type I misclassification rate of the best model in this section was 28.09 per cent and the type II misclassification rate was 22.22 per cent. This translated into an effectiveness of 77.78 per cent and an efficiency of 18.12 per cent. For example, following model recommendations, re-engineering reviews would examine over three-quarters of the fault-prone modules, but less than one in five reviews would find a bona fide re-engineering candidate.

Table 8 shows details of the estimated parameters of each model based on a *fit* data set. Table 9 shows misclassification rates for the product/process metrics-based models. For example, the rates for $test_1$ were a type I misclassification rate of 27.53 per cent and a type II misclassification rate of 25.93 per cent. This translated to an effectiveness of 74.07 per cent and an efficiency of 17.70 per cent. One can expect similar accuracy when classifying modules from a subsequent release or a similar project at the time of release.

Table 8. Models based on product and process metrics

	fit_1		fit_2		fit_3	
Variable	Coefficient	Standard deviation	Coefficient	Standard deviation	Coefficient	Standard deviation
Intercept	−4.9780	0.3398	−4.9636	0.3352	−5.4538	0.3625
FINLINCUQ	+0.0165	0.0052	+0.0146	0.0052	+0.0263	0.0053
LGPATH	+0.0137	0.0086	+0.0136	0.0088	+0.0296	0.0084
VARSPNMX	—	—	+0.00031	0.00019	—	—
USAGE	+1.0921	0.3518	+1.0408	0.3415	+1.0718	0.3617
UNQ_DES	+0.3940	0.0745	+0.2839	0.0530	+0.2862	0.0547
BETA_PR	+0.9057	0.2719	+0.7250	0.2720	+0.4706	0.2147
BETA_FIX	+0.4845	0.1921	+0.4315	0.1839	+0.3217	0.2013
CUST_FIX	—	—	—	—	—	—
VLO_UPD	−0.3958	0.1359	−0.1922	0.1295	−0.3447	0.1290
SRC_GRO	+0.00039	0.00026	—	—	+0.00039	0.00027
UPD_CAR	−0.00037	0.00025	—	—	—	—

— means variable was not significant at 15 per cent level.

Table 9. Evaluation of models based on product and process metrics, $c = 15.98$. Validation by data splitting. Misclassification rates

		fit		*test*	
Data		Type I	Type II	Type I	Type II
All observations		28.09%	22.22%		
Data set	1	27.10%	22.96%	27.53%	25.93%
Data set	2	27.24%	23.53%	25.62%	27.61%
Data set	3	25.28%	22.06%	25.38%	32.84%

Data set	Effectiveness	Efficiency
$test_1$	74.07%	17.70%
$test_2$	72.39%	18.42%
$test_3$	67.16%	17.46%

3.5. Discussion

Owing to the availability of independent variables, the product metrics-based models can be used to guide reviews and testing when coding is complete. Because the number of beta test problems (*BETA_PR*) is available only at the end of beta testing, the product/process metrics-based models can be applied at the time of release to guide planning of support during operations or planning of re-engineering during the next maintenance cycle.

Suppose we conduct a re-engineering review of all the modules identified as *fault-prone* by the product/process metrics-based model. The *test* data sets of the product/process metrics-based models had an average effectiveness of 71.2 per cent and an average efficiency of 17.9 per cent.

If one followed the recommendation of the model, reliability enhancement efforts would be cost-effectively targeted if the value of reducing faults in 71.2 per cent of the *fault-prone* modules is worth more than $5.6 = (1/0.179)$ times the cost of their reviews. Considering the high cost of field problems in telecommunications systems, this factor appears conservative. Therefore, applying such a model would be highly cost-effective.

The product metric-based models had only a few significant variables out of a large set of candidates. This implies faults discovered by customers were found in modules with only a few distinctive product characteristics. Faults in other modules are discovered and corrected prior to release. However, one should note that each significant variable is associated with a different phase of the life cycle. Because the number of distinct include files (*FILINCUQ*) is related to interfaces, it is strongly influenced by high-level design. The logarithm of the number of paths (*LGPATH*) is an attribute of the control flow-graph and thus, is determined by algorithm design. Coding practices influence the usage of variables, and therefore, maximum span of variables (*VARSPNMX*) is determined during the coding phase. *USAGE* approximates the profile of planned use of features which are implemented during the operations phase.

The number of distinct include files (*FILINCUQ*) was the most significant product metric. Include files are often used to implement interfaces among different parts of the software, such as shared constant definitions, shared data types and function prototypes. A large number of include files means that a module has a wide variety of interfaces; this could imply more risk of interface-related mistakes.

EMERALD measures the number of independent paths (*PTHIND*) for each procedure, and then sums over procedures in each module. Since the number of paths at the procedure level can vary over many orders of magnitude, one intricate procedure can dominate the module-level metric. Thus, high values of the module-level number of paths metric (*PTHIND*) may approximate the size of the largest control flow-graph of a procedure. *PTHIND* had some values that were too large for statistical analysis (greater than 10^{32}). A logarithmic transformation converted it to a practical range.

The logarithm of independent paths, $LGPATH = \log_2 PTHIND$, was a significant variable in all of the models. Note that one additional `if-then-else` clause, in sequence with the rest of a module, can double the number of independent paths but only adds 1 to *LGPATH*. Such a minor change in the control flow does not have a large effect on the odds of being fault-prone, even though the number of paths increases dramatically.

The maximum span of variables (*VARSPNMX*) was significant in the models based on data set 2, and in the best models whose *fit* data sets also included the observations in data set 2. EMERALD measures the number of lines of code between the first and last use of a variable within a procedure, which is then aggregated to the module level. This implies that mistakes are more likely when the appearance of variables is less localized.

The deployment score, *USAGE*, was always significant. *USAGE* has a range from zero to one, indicating the deployment proportion of a module among all potential usage sites. Presumably, if a module is deployed, it is also used. *USAGE* is a preliminary approximation of operational usage. In contrast, an ideal execution profile measures the probability of execution (Musa, 1993). In spite of being approximate, *USAGE* was significantly related to operational faults—i.e., those discovered by customers. The fault-discovery process is likely related to the frequency of execution during operations, which is indirectly related to deployment usage. Future research will investigate this relationship more precisely.

The number of different designers that made changes (*UNQ_DES*) was the most significant process metric. Anecdotal evidence indicates that this may be due to two aspects of the maintenance process in this organization. First, work assignments to correct faults are often short term, with little consideration of a programmer's experience with a module. A module with a history of many faults may have been modified by many programmers at various times. Second, 'feature interaction' is a major concern in the telecommunications industry. Many problems discovered by customers consist of undesirable behaviour due to interactions among features that were not anticipated in the requirements. Therefore, a module that is involved with multiple features has more risk of unanticipated interactions. Moreover, maintenance assignments tend to be by feature rather than by module. A given module may be involved in multiple system features, and therefore, all the programmers working on those features will have a need to modify its source code.

The number of beta-test problems (*BETA_PR*), the number of fixes of prior customer problems (*CUST_FIX*), and the number of fixes of prior beta-test problems (*BETA_FIX*) were generally significant variables. *BETA_PR* is a count of faults discovered during beta test, prior to release. Fixing them is an urgent matter; other faults may remain undiscovered in the same module. The faults represented by *BETA_FIX* and *CUST_FIX* were largely discovered in a prior release, and were fixed in the current release. Thus, in general, modules with faults discovered in the field recently are likely to have even more faults discovered after release.

Updates by designers with few prior updates on record (*VLO_UPD*) is a surrogate variable for the experience level of the designers who made changes. Total company-career updates by designers who updated the current module (*UPD_CAR*) embodies a related concept. Since the coefficients of *VLO_UPD* and *UPD_CAR* were negative, updates by inexperienced designers were associated with *not fault-prone* modules. This is counter to the common-sense assumption that less experience implies more mistakes. This phenomenon is explained by the fact that this development organization has strong mentoring of new designers. It is likely that most updates by inexperienced designers were reviewed by their mentors, resulting in fewer faults.

4. CONCLUSIONS

Telecommunications systems are known for high reliability. Their failures can cause major disruptions because society relies on these systems for many important services. Development of reliable software one release after another has become increasingly important in the telecommunications industry because software is the medium for implementing increasingly sophisticated features. Enhanced Measurement for Early Risk Assessment of Latent Defects (EMERALD) is a sophisticated system of decision support tools used to assess risk and to predict software faults at Nortel, which is a major telecommunications systems manufacturer.

The contribution of this industrial case study paper is its demonstration that modules that have faults discovered by customers can be identified with useful accuracy in advance of such a discovery. A small fraction of the modules in the case study had faults discovered by customers (less than 8 per cent). This small set of modules can be difficult to identify. Models were developed for use at the end of the coding phase and the end of beta testing. Both product and process metrics were investigated as independent variables. The set of process metrics in the case study was more extensive than that used in previous research (Khoshgoftaar *et al.*, 1996a). The case study examined a very large legacy telecommunications system with strict reliability requirements. Logistic regression in conjunction

with a generalized classification rule (Khoshgoftaar and Allen, 1997d) predicted whether each module was fault-prone or not. We anticipate that refinements of such models will be incorporated into EMERALD.

Future research may address the following issues. A set of models with other independent variables according to their availability at different points in the life cycle would be useful in the context of systems like EMERALD. Specialized models for important subsystems might have improved accuracy. Model refinements to deal with outliers, interaction among independent variables and sensitivity of models to individual independent variables also could improve accuracy and robustness. When data from a subsequent release become available, another validation study would reinforce the results presented here.

APPENDIX A. LOGISTIC REGRESSION

Logistic regression is a statistical modelling technique where the dependent variable has only two possible values. We find that Hosmer and Lemeshow (1989) is an excellent text for logistic regression. Independent variables may be categorical, discrete, or continuous. In software quality modelling, we usually consider a module as an 'observation'. Our dependent variable is *Class*, which has only two possible values: a module is a member of one group or the other. In this paper, we use the groups not *fault-prone* and *fault-prone*; other groups may be used in other circumstances. Let x_j be the jth independent variable, and let $\mathbf{x}_i$ be the vector of the ith module's independent variable values, for example, software product and process metrics.

We designate a module being *fault-prone* as an 'event'. Let $\hat{p}$ be the probability of an event, and thus, $\hat{p}/(1-\hat{p})$ is the odds of an event. The logistic regression model has the form

$$\ln\left(\frac{\hat{p}}{1-\hat{p}}\right) = \beta_0 + \beta_1 x_1 + \cdots + \beta_j x_j + \cdots + \beta_m x_m \tag{5}$$

where ln means natural logarithm and m is the number of independent variables. Let b_j be the estimated value β_j.

Logistic regression assumes that the probability of an event has a multivariate exponential form:

$$p = \frac{\exp(\beta_0 + \beta_1 x_1 + \cdots + \beta_m x_m)}{1 + \exp(\beta_0 + \beta_1 x_1 + \cdots + \beta_m x_m)} \tag{6}$$

This form is appropriate when the probability is a monotonic function of each individual independent variable. Fortunately, software engineering measures usually do have a monotonic relationship with *Faults* that is inherent in the underlying processes (Basili, Briand and Melo, 1996).

Given a list of candidate independent variables and a threshold significance level, some of the estimated coefficients may not be significantly different from zero. Such variables should not be included in the final model. The process of choosing significant independent variables is called 'model selection'. Stepwise logistic regression is one method of model selection that uses the procedure summarized below (Hosmer and Lemeshow, 1989).

Initially, estimate a model with only the intercept. Evaluate the significance of each variable not in the model. Add to the model the variable with the largest chi-squared p-value which is better than a given threshold significance level. Estimate parameters of the new model. Evaluate the

significance of each variable in the model. Remove from the model the variable with the smallest chi-squared p-value whose significance is worse than a given threshold significance level. Repeat until no variables can be added or removed from the model. Tests for adding or removing a variable are based on an adjusted residual chi-squared statistic for each variable, comparing models with and without the variable of interest.

Maximum likelihood estimates of the parameters of the model, b_j, are calculated using the iteratively reweighted least squares algorithm. Other algorithms are also available to calculate maximum likelihood estimates. This algorithm is used both in the stepwise procedure and for the final model. The estimated standard deviation of a parameter can be calculated, based on the log-likelihood function (Myers, 1990).

An observation, $\mathbf{x}_i$, can be classified by the following procedure: Calculate $\hat{p}/(1-\hat{p})$ using

$$\ln\left(\frac{\hat{p}}{1-\hat{p}}\right) = b_0 + b_1 x_1 + \cdots + b_j x_j + \cdots + b_m x_m \tag{7}$$

Assign the observation by a classification rule, such as the generalized classification rule discussed in Appendix B (Khoshgoftaar and Allen, 1997c, d),

$$Class(\mathbf{x}_i) = \begin{cases} G_1 & \text{if } \dfrac{1-\hat{p}}{\hat{p}} \geq c \\ G_2 & \text{otherwise} \end{cases} \tag{8}$$

where c is chosen experimentally, and G_1 and G_2 are the two groups.

Logistic regression coefficients indicate the marginal influence of an independent variable on the probability of group membership. Large coefficients also lead one to hypothesize cause and effect relationships due to software engineering processes, or to look for common root causes.

APPENDIX B. A GENERALIZED CLASSIFICATION RULE

This Appendix presents a classification rule, which we have proposed for use with software quality models (Khoshgoftaar and Allen, 1997d). It predicts the class of the ith module, $Class(\mathbf{x}_i)$. Table 10 summarizes notation.

Let G_1 be the group of *not fault-prone* modules and G_2 be the group of *fault-prone* modules. Suppose a software quality modelling technique produces a likelihood function, f_k, for each class, $k = 1, 2$. For example, in logistic regression, each likelihood function is a probability of an event, $f_2 = p$, or non-event, $f_1 = 1 - p$. Let p be the probability that a module is *fault-prone*, and let $\hat{p}$ be a logistic regression model's estimate of p. From a Bayesian viewpoint, the class proportions of the population are known prior to applying a model—i.e., the prior probabilities of class membership, π_1 and π_2.

In a standard development process, the expected proportion of *fault-prone* modules is π_2. Following the model's recommendation, the proportion of modules that receive reliability enhancement and that are actually *fault-prone* is $\Pr\{2|2\}\pi_2$. Let us define *effectiveness* as the proportion of *fault-prone* modules that received reliability enhancement treatment out of all the *fault-prone* modules. Then

$$Effectiveness = \frac{\Pr\{2|2\}\pi_2}{\pi_2} = 1 - \Pr\{1|2\} \tag{9}$$

Table 10. Notation

G_1:	the *not fault-prone* class (group) of modules.
G_2:	the *fault-prone* class (group) of modules.
k:	index of classes.
i:	index of modules.
$\mathbf{x}_i$:	the vector of independent variables of the ith module.
$Class(\mathbf{x}_i)$:	the ith module's class, predicted by a model's classification rule.
$f_k(\mathbf{x}_i)$:	a likelihood function for the ith module's membership in the kth class.
p:	the probability that a module is *fault-prone*. In a logistic regression context, $f_2(\mathbf{x}_i) = p_i$.
π_k:	the prior probability of membership in G_k.
Pr{1|1}:	the probability that a model correctly classifies a module as in G_1.
Pr{2|2}:	the probability that a model correctly classifies a module as in G_2.
Pr{2|1}:	the probability that a model misclassifies a module as in G_2 which is actually in G_1, i.e., the type I misclassification rate.
Pr{1|2}:	the probability that a model misclassifies a module as in G_1 which is actually in G_2, i.e., the type II misclassification rate.
C_{I}:	the cost of a type I misclassification.
C_{II}:	the cost of a type II misclassification.

This means that we can maximize *effectiveness* by minimizing Pr{1|2}.

When we apply a reliability enhancement process to a *not fault-prone* module, it will probably be a waste of time, because the reliability is already satisfactory. Let us define *efficiency* as the proportion of reliability enhancement effort that is not wasted. This is equivalent to the proportion of *fault-prone* modules that received reliability enhancement treatment out of all modules that received it:

$$Efficiency = \frac{\Pr\{2|2\}\pi_2}{\Pr\{2|1\}\pi_1 + \Pr\{2|2\}\pi_2} \tag{10}$$

This means that we can maximize *efficiency* by minimizing Pr{2|1}.

We have observed a trade-off between effectiveness and efficiency. As Pr{1|2} goes down, Pr{2|1} goes up, and conversely. Our goal is to choose a practical, flexible classification rule that allows appropriate emphasis on effectiveness and efficiency according to the needs of the project. The following rule enables a project to select its preferred balance between the misclassification rates, and consequently, between effectiveness and efficiency by choosing a constant c:

$$Class(\mathbf{x}_i) = \begin{cases} G_1 & \text{if } \dfrac{f_1(\mathbf{x}_i)}{f_2(\mathbf{x}_i)} \geq c \\ G_2 & \text{otherwise} \end{cases} \tag{11}$$

Applied to logistic regression, this becomes

$$Class(\mathbf{x}_i) = \begin{cases} G_1 & \text{if } \dfrac{1-\hat{p}}{\hat{p}} \geq c \\ G_2 & \text{otherwise} \end{cases} \tag{12}$$

Given a candidate value of c, we estimate misclassification rates Pr{2|1} and Pr{1|2} by resubstitution of the *fit* data set into the model. If the balance is not satisfactory, we select another candidate value of c and estimate again, until we arrive at the best c for the project.

If one chooses c such that $\Pr\{2|1\} = \Pr\{1|2\}$, then the greater of Pr{2|1} and Pr{1|2} is minimized (Seber, 1984). In effect, this choice of c minimizes both misclassification rates. In practice, we can achieve only approximate equality due to finite discrete data sets.

One can interpret the preferred value of c in the context of a classification rule that minimizes the expected cost of misclassifications (Khoshgoftaar and Allen, 1997b). In software engineering practice, the penalty for a type II misclassification is often much more severe than for a type I. A reliability enhancement technique, such as extra reviews, typically has modest direct cost per module, C_{I}. On the other hand, the cost of a type II misclassification, C_{II}, is the lost opportunity to correct faults early. The consequences of letting a fault go undetected until after release can be very expensive. Thus, a classification rule should take into account the costs of each kind of misclassification.

The expected cost of misclassification (*ECM*) of one module is

$$ECM = C_{\mathrm{I}} \Pr(2|2)\pi_1 + C_{\mathrm{II}} \Pr(1|2)\pi_2 \tag{13}$$

A classification rule that minimizes the expected cost of misclassification (Johnson and Wichern, 1992; Seber, 1984) is

$$Class(\mathbf{x}_i) = \begin{cases} G_1 & \text{if } \dfrac{f_1(\mathbf{x}_i)}{f_2(\mathbf{x}_i)} \geq \left(\dfrac{C_{\mathrm{II}}}{C_{\mathrm{I}}}\right)\left(\dfrac{\pi_2}{\pi_1}\right) \\ G_2 & \text{otherwise} \end{cases} \tag{14}$$

The generalized classification rule does not depend on our knowledge of π_1 and π_2, nor of C_{I} and C_{II}. If such information is available, having selected a preferred c, one can interpret the value of c in terms of the prior probabilities of class membership ratio times the cost ratio, as in the minimum expected cost rule:

$$c = \left(\frac{C_{\mathrm{II}}}{C_{\mathrm{I}}}\right)\left(\frac{\pi_2}{\pi_1}\right) \tag{15}$$

Acknowledgements

We thank the EMERALD and Dev Metrics teams at Nortel for collecting the data, and the Nortel Software Reliability Department at Research Triangle Park in North Carolina for its sponsorship. We thank the *Journal's* referees for their helpful comments. The findings and opinions in this paper belong solely to the authors, and are not necessarily those of the sponsor. Moreover, our results do not in any way reflect the quality of the sponsor's software products.

References

Basili, V. R., Briand, L. C. and Melo, W. (1996) 'A validation of object-oriented design metrics as quality indicators', *IEEE Transactions on Software Engineering*, **22**(10), 751–761.

Briand, L. C. and Basili, V. R. (1992) 'A classification procedure for the effective management of changes during the maintenance process', in *Proceedings Conference on Software Maintenance*, IEEE Computer Society Press, Los Alamitos CA, pp. 328–336.

Coleman, D., Lowther, B. and Oman, P. (1995) 'The application of software maintainability models in industrial software', *Journal of Systems and Software*, **29**(1), 3–16.

Dillon, W. R. and Goldstein, M. (1984) *Multivariate Analysis: Methods and Applications*, John Wiley & Sons, Inc., New York NY, 587 pp.

Ebert, C. (1997) 'Experiences with criticality predictions in software development', in *Software Engineering—ESEC/FSE '97: 6th European Software Engineering Conference and the 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Proceedings, vol. 1301 of Lecture Notes in Computer Science*, Springer-Verlag, Berlin, pp. 278–293. Also published in *ACM SIGSOFT Software Engineering Notes*, **22**(6), 278–293.

Henry, S. and Wake, S. (1991) 'Predicting maintainability with software quality metrics', *Journal of Software Maintenance*, **3**(3), 129–143.

Hosmer, Jr., D. W. and Lemeshow, S. (1989) *Applied Logistic Regression*, John Wiley & Sons, Inc., New York NY, 307 pp.

Hudepohl, J. P. (1990) 'Measurement of software service quality for large telecommunications systems', *IEEE Journal of Selected Areas in Communications*, **8**(2), 210–218.

Hudepohl, J. P., Aud, S. J., Khoshgoftaar, T. M., Allen, E. B. and Mayrand, J. (1996) 'EMERALD: software metrics and models on the desktop', *IEEE Software*, **13**(5), 56–60.

Hudepohl, J. P., Snipes, W., Hollack, T. and Jones, W. (1992) 'A methodology to improve switching system software service quality and reliability', in *Proceedings IEEE Global Telecommunications Conference*, IEEE Communications Society, New York NY, pp. 1671–1678.

Johnson, R. A. and Wichern, D. W. (1992) *Applied Multivariate Statistical Analysis*, 3rd edition, Prentice-Hall, Englewood Cliffs NJ, 642 pp.

Khoshgoftaar, T. M. and Allen, E. B. (1997a) 'Classification techniques for predicting software quality: lessons learned', in *Proceedings Annual Oregon Workshop on Software Metrics*, University of Idaho, Moscow ID, track 2, paper 1.

Khoshgoftaar, T. M. and Allen, E. B. (1997b) 'The impact of costs of misclassification on software quality modelling', in *Proceedings Fourth International Software Metrics Symposium*, IEEE Computer Society, Los Alamitos CA, pp. 54–62.

Khoshgoftaar, T. M. and Allen, E. B. (1997c) *Logistic regression modeling of software quality*, Technical Report TR-CSE-97-24, Florida Atlantic University, Boca Raton FL, 18 pp.

Khoshgoftaar, T. M. and Allen, E. B. (1997d) *A practical classification rule for software quality models*, Technical Report TR-CSE-97-56, Florida Atlantic University, Boca Raton FL, 27 pp.

Khoshgoftaar, T. M., Allen, E. B., Halstead, R., Trio, G. P. and Flass, R. (1998) 'Process measures for predicting software quality', *Computer*, **31**(4), 66–72.

Khoshgoftaar, T. M., Allen, E. B., Kalaichelvan, K. S. and Goel, N. (1996a) 'Early quality prediction: a case study in telecommunications', *IEEE Software*, **13**(1), 65–71.

Khoshgoftaar, T. M., Allen, E. B., Kalaichelvan, K. S. and Goel, N. (1996b) 'Predictive modelling of software quality for very large telecommunications systems', in *Proceedings International Communications Conference*, volume 1, IEEE Communications Society, New York NY, pp. 214–219.

Lyu, M. R., Yu, J. S., Keramidas, E. and Dalal, S. R. (1995) 'ARMOR: analyzer for reducing module operational risk', in *Twenty-Fifth International Symposium on Fault-Tolerant Computing, Digest of Papers*, IEEE Computer Society Press, Los Alamitos CA, pp. 137–142.

Mayrand, J. and Coallier, F. (1996) 'System acquisition based on software product assessment', in *Proceedings of the Eighteenth International Conference on Software Engineering*, IEEE Computer Society Press, Los Alamitos CA, pp. 210–219.

Musa, J. D. (1993) 'Operational profiles in software reliability engineering', *IEEE Software*, **10**(2), 14–32.

Myers, R. H. (1990) *Classical and Modern Regression with Applications*, Duxbury Series, PWS-KENT Publishing, Boston MA, 488 pp.

Robillard, P. N., Coupal, D. and Coallier, F. (1991) 'Profiling software through the use of metrics', *Software—Practice and Experience*, **21**(5), 507–518.

Russell, G. W. (1991) 'Experience with inspection in ultralarge-scale developments', *IEEE Software*, **8**(1), 25–31.

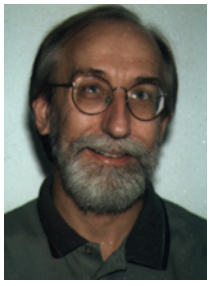
Schneidewind, N. F. (1995) 'Controlling and predicting the quality of space-shuttle software using metrics', *Software Quality Journal*, **4**(1), 49–68.
Seber, G. A. F. (1984) *Multivariate Observations*, John Wiley & Sons, Inc., New York NY, 686 pp.

Authors' biographies:

Taghi M. Khoshgoftaar is a Professor of the Department of Computer Science and Engineering, Florida Atlantic University, and the Director of the Empirical Software Engineering Laboratory. His research interests are in software engineering, software complexity metrics and measurements, software reliability and quality engineering, computational intelligence applications, computer performance evaluation, multimedia systems, and statistical modelling. He received an M.S. in Applied Mathematics from Massachusetts Institute of Technology, an M.S. in Computer Science from North Carolina State University, and a Ph.D. in Statistics from Virginia Polytechnic Institute and State University. His email address is taghi@cse.fau.edu

Edward B. Allen is a Research Associate and an adjunct Professor in the Department of Computer Science and Engineering at Florida Atlantic University. His research interests include software measurement, software process modelling and software quality. He received a B.S. in Engineering from Brown University in Providence RI in 1971, an M.S. in systems engineering from the University of Pennsylvania, Philadelphia PA in 1973, and a Ph.D. in computer science from Florida Atlantic University at Boca Raton FL, in 1995. His email address is edward.allen@computer.org

Wendell D. Jones received his B.S. in Mathematics from Furman University at Greenville SC in 1983 and his M.S. and Ph.D. in Mathematical Science (Statistics emphasis) from Clemson University at Clemson SC in 1985 and 1987, respectively. He has held various positions in his 11 years with Nortel, most recently as Advisor and Chief Researcher of the Department of Software Reliability Engineering and Tool Development. He is an active member of the American Statistical Association and has served on several technical committees related to software reliability. His research interests include applying quantitative methods to software development and management, and applications of data analysis to large data sets. His email address is wendellj@nortel.ca



John P. Hudepohl is currently Manager of Software Reliability Engineering at Nortel, Research Triangle Park NC in the USA. He has more than 20 years experience in the reliability, maintainability and quality fields for both hardware and software. He received a B.S. in both Electronic Engineering and Communication Arts from the University of Dayton, and an M.S. in System Management from Florida Institute of Technology. His email address is hudepohl@nortel.ca

